The Vision Egg Programmer's Manual

Release 1.1.1

Andrew Straw

September 7, 2008

Abstract

The Vision Egg was designed to perform two primary tasks. The first task is the drawing of computer graphics using OpenGL. The optional second task is to handle the flow control of your program to coordinate events on your computer in a precisely timed way.

These are challenging tasks, and the Vision Egg does much of the work for you. However, to make full use of the Vision Egg, you should understand the basics. This is an overview of the main components of the VisionEgg itself.

Note before starting: The Vision Egg is fundamentally object oriented in nature, and this document assumes you are familiar with terms such as ”class” and ”instance”. If you are not, please find some information on the topic of object oriented programming. As you write scipts, you will also need to consult the Python, Numeric, pygame, or other documentation.

CONTENTS

CHAPTER

ONE

Coordinating events

There are several ways to organize the sequence of your experiments using the Vision Egg. You can write your own custom flow control and event handling, using the Vision Egg solely for drawing graphics. This is often useful in psychophysics experiments where interaction with a subject is interleaved with presentation of stimuli.

Alternatively, you can make use of the classes in `VisionEgg.FlowControl`. For example, `Presentation` is a class that maintains an association between the parameters of stimuli and their control functions, calls these functions, and initiates drawing of the stimuli. There are several ways to use the `Presentation` class described below. This mode of operation is useful for electrophysiology experiments.

1.1 Custom flow control and event handling

By writing your own custom flow control code, you have much more flexibility in designing experiments, but also less of the work involved has been done for you. Perhaps the best place to start is simply to look at some examples. See the demonstration scripts dots_simple_loop.py, mouseTarget_user_loop.py, and multi_stim.py. Each of these programs has its own main loop which performs the same role as the `Presentation` class's `go` and `run_forever` methods, which are described further in this chapter.

1.2 Using the Presentation class: Running a single trial

Most of the Vision Egg demonstration scripts run a single trial and then quit. From a programming perspective, this is the easiest way to get started. The timing and control of events within a single trial is performed by a "go loop", which is entered by calling the `go()` method of the `Presentation` class.

A cycle of the go loop consists of updating any relevant stimulus parameters, clearing the framebuffer, and calling the stimuli to draw themselves. The buffers are swapped and the cycle begins again, usually after waiting for the vertical blanking interval (see the section in this manual on double buffering). When waiting for the vertical blanking interval ("sync swap buffers") is enabled, cycles through the "go loop" never occur faster than the frame rate. If the go loop is burdened with lots of calculations or if the operating system takes the CPU away from the Vision Egg, the cycle through the go loop is not completed before the video card begins drawing the next frame and therefore a frame is skipped.

A go loop can run indefinitly or have its duration limited to a duration measured in seconds or in number of frames drawn. (Measuring duration based on frames drawn is only meaningful when buffer swapping is synchronized with the vertical blanking interval and frame skipping would be particularly undesirable in this case.)

1.3 Using the Presentation class: Continuous operation

Often, the visual stimulus needs to continue running between trials. At a minimum this could be used to keep the display constant and to prevent the Vision Egg from quitting, but could also be used to maintain a moving pattern on the display between trials. In addition, it may be necessary to trigger a go loop with a minimum of latency after the receipt of some signal, such as a digital input on the parallel port.

To use the Vision Egg in this manner, the `run_forever()` method of `Presentation` is called, which begins a loop that performs the same tasks as a go loop with the exception that functions controlling stimulus parameters are informed that it is a "between go loops" state. At any point this `run_forever` loop can create a go loop, which returns control back to the `run_forever` loop when done. Alternatively, if the controlling functions for stimulus parameters operate between go loops, the entire experiment could be run without entering a go loop. (This could also be acheived by starting a go loop with a duration parameter set to "forever".)

CHAPTER

TWO

Hierarchy of graphical objects

Currently, the Vision Egg supports only a single screen (window). However, it is designed to run simultaneously in multiple screens, so once this capability is available (perhaps in pyglet), the following priciples will continue to apply.

Each screen contains a list of least one "viewport". A viewport is defined to occupy a rectangular region of the screen and define how and where objects are drawn. The default viewport created with each screen fills the entire screen. In the Vision Egg `Viewport` class, the screen position and size are specified in addition to the projection. The projection, specified by the `Projection` class, transforms 3D "eye coordinates" into "clip coordinates" according to, for example, an orthographic or perspective projection. (Eye coordinates are the 3D coordinates of objects referenced from the observers eye in arbitrary units. Clip coordinates are used to compute the final position of the 3D object on the 2D screen.) The default `Projection` created with a `Viewport` is an orthographic projection that maps eye coordinates in a one to one manner to pixel coordinates, allowing specification of object position in absolute pixels. For more information, consult section 2.11, "Coordinate Transformations" of the OpenGL Specification.

Multiple instances of the `Viewport` class may occupy the same region of the screen. This could be used, for example, to overlay objects with different projections such as in the targetBackground demo. The order of the list of viewports is important, with the first in the list being drawn first and later viewports are drawn on top of earlier viewports.

An instance of the `Viewport` class keeps an ordered list of the objects it draws. Objects to be drawn on top of other objects should be drawn last and therefore placed last in the list.

The objects a viewport drawns are all instances of the `Stimulus` class. The name "Stimulus" is perhaps a slightly inaccurate because instances of this class only define how to draw a set of graphics primitives. So for example, there are `SinGrating2D` and `TextureStimulus` subclasses of the `Stimulus` class.

The Vision Egg draws objects in a hierarchical manner. First, the screen(s) calls each of its viewports in turn. Each viewport calls each of its stimuli in turn. In this way, the occlusion of objects can be controlled by drawing order without employing more advanced concepts such as depth testing (which is also possible).

CHAPTER

THREE

Controlling stimulus parameters in realtime

When using the `Presentation` class, you have a powerful method of updating parameters in realtime available to you. "Controllers" are instances of the class `Controller`. A controller is called at pre-defined intervals and updates the value of some stimulus parameter. For example, in the "target" demo script, the "center" parameter of a `Target2D` stimulus is updated on every frame by a function which computes position based upon the current time. You can also control parameters without using controllers by simply changing the values as your program executes.

Instances of `Controller` are called by instances of the `Presentation` class. After creating an instance of `Controller`, it must be "registered" by calling the `add_controller` method of `Presentation`, during which the stimulus parameter under control is specified. The `Presentation` takes care of calling the controller from this point. Specifically, the `during_go_eval()` is called during a `go()` loop, and `between_go_eval()` is called by `between_presentations()` (during `run_forever()`, for example.) These "eval" methods return a value which becomes the new value of the parameter being controller.

The frequency with which `during_go_eval()` and `between_go_eval()` are evaluated is determined by the *eval_frequency* attribute of the controller. The default *eval_frequency* is every frame.

The *temporal_variables* attribute of the controller specifies what temporal variables the "eval" methods have available to base calculations on. The default value is TIME_SEC_SINCE_GO, so when `during_go_eval()` is called, the instance will have an attribute *time_sec_since_go* set to the time since the onset of the `go()` loop.

For more information, see the documentation for the `Controller` and `Presentation` classes in the `VisionEgg.Core` module.

CHAPTER

FOUR

Other general information

4.1 Double buffering

The Vision Egg operates in double buffered mode. This means that the contents of the "front" framebuffer are read by the video card to produce the on-screen display. Meanwhile, clearing and drawing operations always occur on the back framebuffer, which becomes the front buffer on a buffer swap (also called flip). This way, an incomplete frame is never drawn to the screen. However, if the buffers are swapped while the display is only part-way through the front buffer, the top and bottom parts of the display will contain frames drawn at different times and thus lead to a tearing artifact. For this reason the default behavior of the Vision Egg is to synchronize buffer swapping with the vertical blanking period, ensuring that tearing does not happen. (Synchronizing buffer swapping is not available for some video drivers on the linux platform, and may frequently be overriden in the Displays control panel in Windows.)

4.2 File layout

Several directories are created in a Vision Egg installation. The files used when a Python script imports any Vision Egg module are in the standard Python "site-packages" directory. Next, the Vision Egg system directory contains data files such as sample images used by the demo scripts and a site-wide configuration file. A user-specific configuration file is looked for is the VisionEgg home directory. On your system, you can determine the exact location of these directories by running the "check-config.py" script, which is distributed with the Vision Egg. If you have installed the Vision Egg from source, you will also have the source directory.

4.3 Priority control

The Vision Egg can take advantage of operating system dependent methods of setting the priority of an application. The performance and features vary from platform to platform. The options available from OS specific system calls are available in the Vision Egg. Before trying something new, do not attempt to increase your script's priority, because this may result in locking up the computer.

4.4 The log file: Warnings and errors

The Vision Egg uses the standard Python logging package to log information including warnings and errors to two locations by default: the standard error stream (as standard for Python scripts) and to a file called "VisionEgg.log" in the current directory. The standard error stream is typically printed on the console, which may only be visible when running your script from the command line. If your script (or modules it imports) raise a `SyntaxError`, the Vision Egg will be unable to start and therefore unable to copy the exception traceback to the log file, and viewing the standard error is the only way to see what went wrong. Therefore, if your log file does not display an error but your

script is not executing, run it from the command line. Also, on Mac OS X, the standard error output is only visible through the Console.app available in "/Applications/Utilities".

You can increase the verbosity of the output by doing something like "VisionEgg.logger.setLevel(VisionEgg.logging.DEBUG)" in your script.

4.5 Configuration options

A number of options that control behavior of the Vision Egg are available. These options are first determined when the `VisionEgg.Configuration` module is loaded (by the base module `VisionEgg`, for example). This module first loads variables from the "VisionEgg.cfg" file and then checks for environment variables that override these values.

The values set by `VisionEgg.Configuration` may be overriden at any time by re-assigning the appropriate variable. For example:

```
import VisionEgg

# Turn off GUI window when calling create_default_screen()
VisionEgg.config.VISIONEGG_GUI_INIT = 0
```

4.6 For more information

A significant amount of documentation is contained within the source code as "docstrings" — special comments within the code. These docstrings are often the best source of information for a particular class or function. In particular, the fundamentally important classes in the `VisionEgg.Core` module are well documented. You can either browse the source code itself, look at the library reference compiled from the source, or use a utility such as PyDoc to compile your own reference from the source.

The Vision Egg mailing list is another source of valuble information. Sign up and browse the archives through the Vision Egg website.

For installation instructions, the Vision Egg website provides the most up-to-date, platform-specific information.

To create your own stimuli you need to know OpenGL. To learn more about OpenGL, you may want to begin with "The Red Book" (The OpenGL Programming Guide, The Official Guide to Learning). The OpenGL specification is also useful (available online).